EXHIBIT 31

Document Number	ENG-54492
Revision	A
Author	Marco Di Benedetto, Shawn Yang
Project Manager	Ramana Mellacheruvu

IOS/ENA Spanning Tree Core

Software Unit Functional/Design Spec

Project Headline

This document specifies all the functional and design aspects of the STP core. In this document we won't discuss any platform dependent issue.

Reviewers

Department	Name	Acceptance Date
020020950 (Dept Number)	Umesh Mahajan	mm/dd/yy

Modification History

Rev.	Date	Originator	Comment
A	2/24/00	Marco Di Benedetto	First draft

Definitions

This section defines words, acronyms, and actions which may not be readily understood.

STP	Spanning Tree Protocol.
PVST	Per VLAN Spanning Tree.
MISTP	Multi Instance Spanning Tree Protocol.
BPDU	Bridge Protocol Data Unit. A packet contains the encoded data as needed by STP computation.
IOS/ENA	Cisco IOS Extended Network Architecture
NetIO	Network Input/Output
Interface Manager	The ENA component responsible for management of interfaces
SysDB	The System Database, which is a key component of IOS/ENA. It servers as a global repository for configuration and other operational information. It provides centralized storage of data, as well as verification and notification functions when this data is created or updated.

System Manager	The IOS/ENA component that is responsible for initially starting up (and restarting when necessary) any components for the router/switch to function.

A printed version of this document is an uncontrolled copy.

A printed version of this document is an uncontrolled copy.

A printed version of this document is an uncontrolled copy.

A printed version of this document is an uncontrolled copy.

1.0 Overview

The purpose of the spanning tree protocol is to provide a single loop-free, logical topology from a meshed redundant physical topology. This document describes how the spanning tree protocol is implemented under IOS/ENA. This includes how the STP algorithm is handled by various processes and components, and how it interacts with other system modules. The discussion in this document focuses on the platform independent part of the implementation of STP. Companion documents are required to fully understand the entire implementation on a specific platform (as an example, the platform dependent portion of the catalyst 6000 STP is explained in [2]). For this reason, here our goal is to achieve the highest platform independence, as well as modularity and scalability.

It is assumed that the reader has an overall understanding of IOS/ENA. No attempt has been made to discuss the specific details about that.

The STP process will implement the Spanning Tree Protocol as specified in the IEEE 802.1D standard ([1]). It will spawn multiple threads to implement a scalable set of STP instances. Each instance may be related to one or more VLANs. No specific assumptions have been made on the number of spanning tree instances, and the way they are related to the VLAN configuration.

2.0 Terminology

There are some words we`ll try to use consistently throughout this document. In this section we are going to give a unique definition for them.

- Instance: the set of data used by STP to determine a logical loop free topology (section 3.1).
- Port: the smallest entity used by STP for the topology computation. A port is an interface applied to an instance (section 3.1).
- STP extension: code interfacing the STP core with the IM interfaces. An extension defines a set of instances and the way ports are added to those instances, while the core provides the functionality to be applied to the defined instances.
- STP features: enhancements applied to the core code. Features are extension independent and augment the functionality of the core.
- Core instance number (CIN): an index in the flat space of the instances as defined by the STP core. Each CIN indexes an entry in the instance table kept by the core.
- Extension handle: a unique number used to reference a specific extension in the core. The core does not know what the extension does, but it uses the handle to index cookies and function pointers registered by the extension.
- Extension instance number (EIN): an index in a "virtual" per-extension instance table. Each extension can address its own instances by using this values (from 0 to a maximum defined by the extension). As the core keeps the only real instance table available, when the number goes to the core, it has to be mapped to a CIN.
- Instance handle: a pair of (extension handle, extension instance number) that allow the core to do the mapping of an EIN to a CIN.

3.0 Functionality

3.1 Architectural overview

The most important element of the STP database is the port (also called "instance-port"). A port is an interface used in an instance, and therefore the STP database could be required to create many ports whenever a single new interface is created; each different port will be referring to the same interface in the context of a specific instance. Given this definition, follows the important property that each interface can be represented in an instance by at most one port.

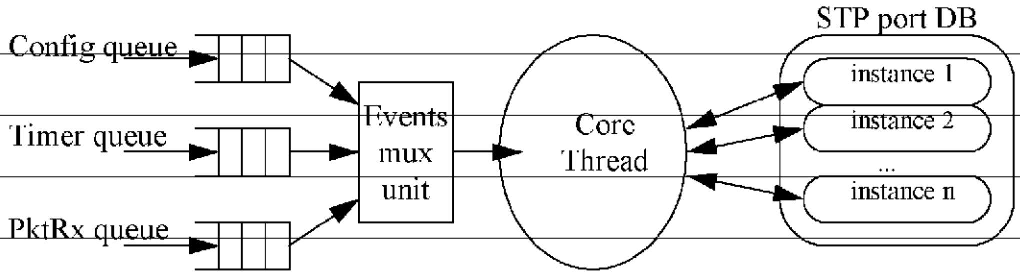

Figure 1, Rough single thread model

An STP instance is a collection of ports participating in the computation of a single logical loop free topology. Every instance is completely self-contained, and therefore there is no necessity for an instance to share information with any other instance. The computation of the topology for each instance uses the algorithm defined by the IEEE standard 802.1D and requires the generation of packets (Bridge Protocol Data Units, BPDUs) on a per port basis and in a timely fashion.

Depending on the STP extension running on a system, the number of instances and ports can become very high. Taking PVST+ as an example, this extension uses a one-to-one relationship between an instance and a VLAN. As up to 4096 VLANs can be defined on a system, the number of ports maintained by the STP can quickly become huge. Moreover, the CPU requirements for the Spanning Tree Protocol grow linearly with the number of ports, therefore the load on the system can rapidly become significant.

For this reasons, the design of the STP core should take scalability as the most important goal to achieve, trying to create a balance with the requirements of the protocol.

The STP process can receive different kind of events:

- Configuration events
- Timer events
- BPDUs

The reception of BPDUs happens at the highest frequency, with frequency decreasing for timer and configuration events, which are the less frequent. To gain responsiveness in the process, we intend to prioritize configuration and timer events over BPDUs reception. Therefore we'll try to maintain three different event queues for the different events priorities (as shown in figure 1).

Although the event priorities would suggest a model with a thread per event queue, given the properties of the STP database model, the natural selection for the computation and update would use a single thread handling completely an instance, and possibly different thread handling different instances. It is not acceptable for the STP core process to spawn one thread per instance, therefore the chosen model uses one single thread responsible for all the instances (see figure 1). The event multiplexing unit is responsible to pass events to the core thread in a prioritized fashion. This means that config events will be processed first, and only when they are not available in the queue, then timer events will be processed; finally, if both the queues are empty, BPDUs will be processed.

This "rough" approach would have many disadvantages; in the next sections we try to tune the model to be more suitable for the specific type of computation the Spanning Tree core protocol requires.

3.2 Refining the single thread model

From the catOS implementation we gained some knowledge of the limitations of the Spanning Tree protocol. The most important issue is related to the necessity of slow blocking calls every time a port state is changed. These calls can have a big impact on the behaviour of the protocol, because of the chain effect they have: during a big topology change many blocking calls can be required, but while the thread is blocking, low

A printed version of this document is an uncontrolled copy

priority events (i.e. BPDUs received) can accumulate on the queue, and processing of other timer events get skewed; therefore other topology changes can be triggered by the delay introduced in the timer and BPDU processing.

To solve this issue, we are going to batch some blocking calls and create another pool of threads servicing the batch (see figure 2). Because of the specific nature of the task requested to this pool, those threads will be waiting most of the time: sometime they will be waiting for a batch of requests from the main thread, some other times they will be waiting for a batch of blocking calls to complete. This thread pool will never have any "active" computation responsibility, and therefore the number of thread in the pool will hold few resources, but it will never impact the performances of the CPU, and furthermore, it will never try to access the port database, thanks to the very specific task requested to it.

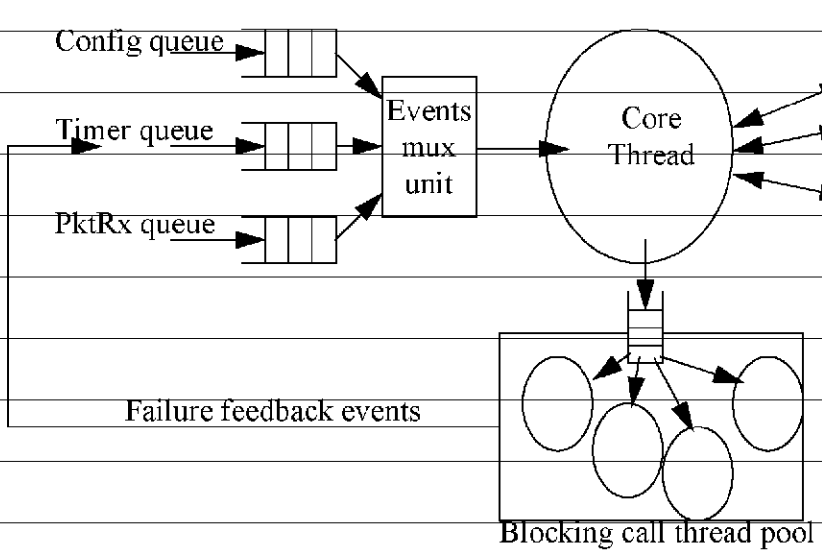

Figure 2. Single non-blocking thread model

We think that the thread pool servicing blocking calls can be optimized on a per platform basis, and therefore in this document we will just give a high level description of the pool's functionality and of the communication scheme chosen to let the main thread creating a batch and be informed in case of failures. The core will get this service though well defined APIs linked into a platform specific DLL.

Just a special note about the events produced by the thread pool as a feedback for failure cases: the overhead introduced by those events should be very small, given the small probability of a failure to happen.

3.3 Trying to optimize further

Removing from the main thread the responsibility to handle slow blocking calls makes the main thread a non-blocking ready-to-answer thread. The thread can therefore react to any incoming event with the maximum speed possible; it also uses a prioritization mechanism in order to react first to critical events. Only after critical events have been serviced, the simple hello messages can be checked, as they *usually* do not carry any amount of new information and can be completed very quickly.

Although the assumption of "simple hello messages" not carrying new information is true in the steady state, a topology change usually invalidates the assumption. When this happens, based on the reception of a superior BPDU the Spanning Tree can trigger a complete recomputation for the entire instance involved.

The computation required by such low priority events can delay higher priority events because low priority events cannot be preempted while under execution. Actually, the entire single thread model is based on the assumption that an event must be processed to completion before another event can be handled. If you think about it however, this assumption strictly covers a requirement only if all the events pertain to the same

A printed version of this document is an uncontrolled copy.

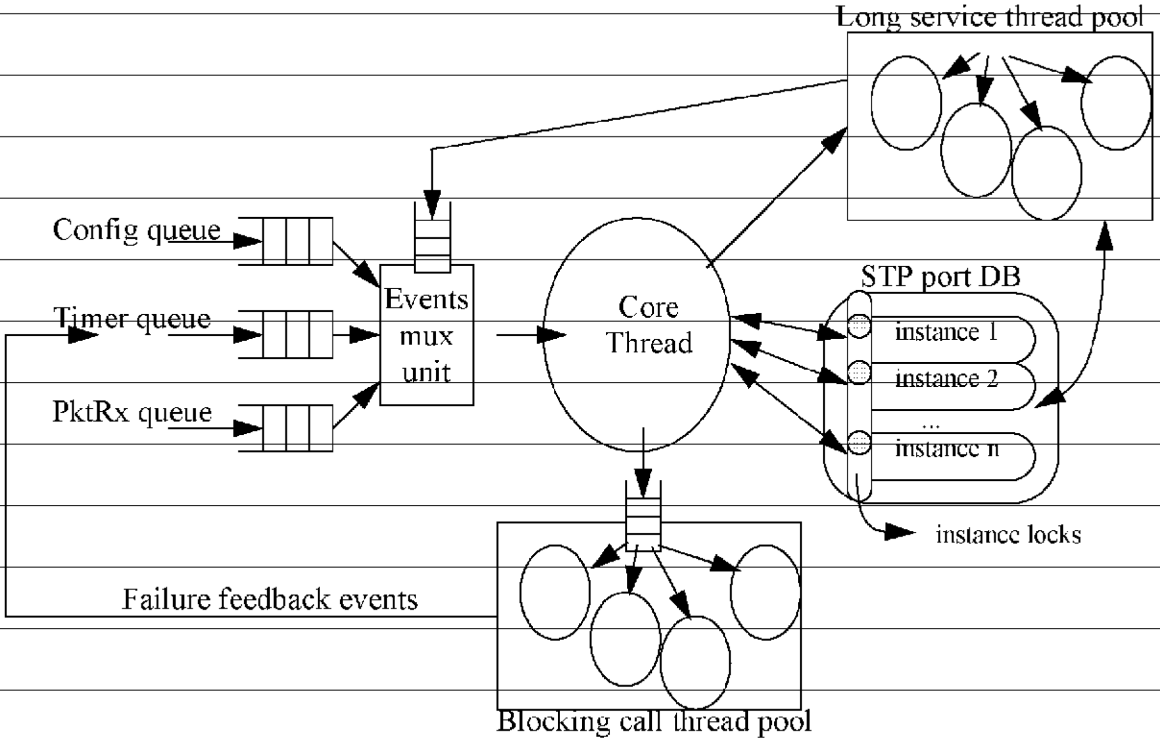

Figure 3, Single non-blocking thread model with "preempted long service" support

instance: in fact events for the same instance must be processed in order to avoid inconsistencies, but the property of each instance being a self-contained object makes this a non-requirement for events in different instances, which could be processed in any order. To use an example, if I receive an event for instance 1 and an event for instance 2 in this order, why am I forced to process them in the same order?

The answer to this question could be "because it is simpler", and provided that the events require a short processing time, it is also not worth to change order or try to make the processing parallel: we have one CPU, and no matter whether we use one thread or two threads, if at time *t* both the events are completed if we process them sequentializing the processing, they would be completed at the same time also if we parallelized the processing (and vice-versa). The total amount of time required to process the two events does not change.

But if we now take into account events which require a long processing time, although it is still true that the total amount of time required to process both does not change, the short event can be subject to an unfair latency because of the processing of the long event. And we know that such a delay could have a chain effect and create instability in the topology of instance 2 (waiting for a short service) just because instance 1 happens to be reconfigured at that time (servicing a long event).

This risk brought us to think about the possibility to move the event requiring a long processing to a new thread, and let the main thread handling only events requiring short processing time. With this optimization we could have another thread pool waiting to service long events (see figure 3); the main thread will have complete control over the processing of those threads and will be in charge to guarantee that events destined to the same instance are subject to a sequentialized processing. For this purpose a simple locking mechanism can be used. Obviously, when all the threads in the pool are busy, the main thread won't wait for them to process another long event and it will process the event itself; in this case all the new events will stay waiting in the queues.

Here we just want to point out that the control on the lock of an instance is completely left to the main thread. If an event comes and the main thread decides it requires a long processing, it will move the processing request to a thread in the pool and lock the instance. When the processing finishes, the signal from the slave thread will

A printed version of this document is an uncontrolled copy.

cause the main thread to unlock the instance. What happens then if the main thread receives a new event for an instance which is locked by a long event under processing? The simple answer would be that in this case only the main thread has to block and wait for the slave thread to return the control of the instance.

An alternative could be to just enqueue the new event on a queue for the instance and let the main thread proceed with the next event. As soon as the slave thread signals that it finished processing the instance, the main thread can unlock it and verify whether the queue of the instance is empty. If that queue is not empty, all the events waiting there have priority over the new events coming, because those events are supposed to have been serviced already, while instead they have been only moved from one queue to another. If another signal comes from another slave thread, the events stored in all the available queues of the instance locks should be serviced in a round robin fashion, and exhausted before any new event coming can be processed.

Finally, if you ask whether this last scheme introduces more overhead to the STP core or not, the answer is that the overhead comes only when the main thread would be supposed to wait, and therefore it causes extra-processing as opposed to non-processing, not extra-processing at the expenses of other events.

Given all this discussion to reduce the latency of high priority events, we won't implement this scheme right now. The reason being that events can easily involve multiple instances and break our initial assumption; just as an example, for optimization purposes a single event could advertise an interface added to more than one instance, when a trunk comes up, as opposed to one event per instance or VLAN. We have chosen to gain this IPC/notification optimization and lose some processing optimization.

3.4 STP timers management

STP makes intensive use of timers. Each STP instance contains three instance timers (hello, TCN, and topology change) and three port timers (forward delay, message age, and hold). Therefore, the total number of timers is proportional to the number of instances and ports. This could be in the order of millions on certain platforms like Catalyst 6000.

Two key observations motivate the timer design.

- There exists a large diversity in the number, usage frequency, and time length in all the timers.
- The usage of many of the STP timers is very low; they are needed for dealing with special cases only.

IOS/ENA offers the tool of managed timer for handling large number of timers. A managed timer maintains a tree of timers in which each intermediate node takes the earliest expiry time of all its children nodes. Hence the root node represents the earliest timer in the entire tree. The leaf nodes are the real control timers that are started and stopped directly. Each level of the tree is maintained as a sorted list. Upon expiry, a notification will be sent to the managed timer owner, and the expired timer can be quickly located by getting the context of the root node.

STP creates a managed timer tree with the hello, TCN, topology change, and forward delay timers as the leaf nodes (figure 4). The timers are organized in the following hierarchy:

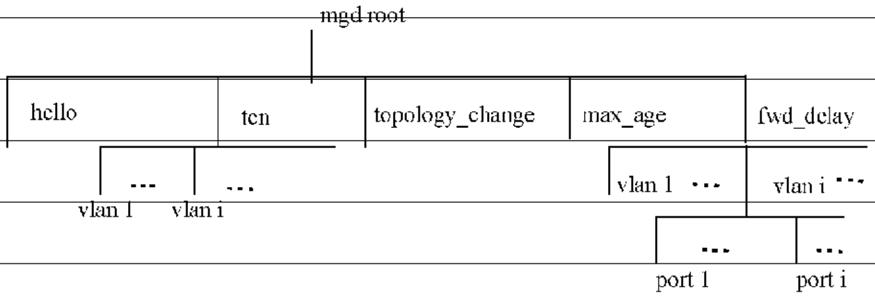

Figure 4, STP managed timers

- Timer type

A printed version of this document is an uncontrolled copy.

- Instance
- Port

The hello timer, TCN timer, and topology change timer are located at the second level (instance) because they are instance timers. Only the forward delay timers are at the third (port) level. The reason for this "instance first" organization is that timer activities are in general more instance-based than port-based. For example, when a topology change is detected in a certain instance, most likely multiple timers in that instance will be affected. This arrangement confines the influence to a smaller region in the tree and reduces the number of insertion sorts needed to restore the order.

3.4.1 Message age timers

Message age timers are not directly put in the managed timer tree because of their extremely high usage. The message age timer is updated for every BPDU received. Having it in the managed timer will cause a considerable sorting overhead related to packet activity. Message age timers have a long time length (15-30 seconds), and hardly expire. Therefore, most of such sorting operations are in fact wasted, since expiry notifications are normally not generated.

Message age timers use a combination of passive timers and managed timers. The purpose is to minimize sorting overhead by using the idea of "counting sort". Specifically, we put all the message age timers in a a queue of timer "buckets" by expiry time. The buckets are created to represent all the time instants from the current time to the largest max age time (30 seconds) in a certain precision (e.g. 1 second). Therefore, there is a bucket to represent the time "right now", and a bucket to represent the time "1 second in the future"…. The last bucket is for "30 seconds in the future". Each bucket keeps a list of timers that expire during that time slice (figure 5). When message age timers are updated, they will be moved from their old buckets to their respective new buckets. STP always maintains the pointer to the first non-empty bucket next to the current bucket. At the time this bucket expires, STP scans the timer list in that bucket and processes the expiry events. Meanwhile, the entire bucket list is shifted circularly by the amount of the slipped time, which means that the expired bucket becomes the bucket for the time "right now", and the one immediately before it becomes that for "30 seconds in the future".

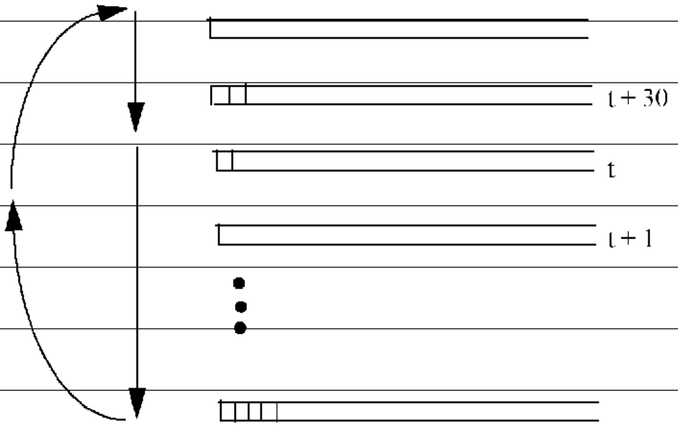

Figure 5. Message age timers organization

Now, because only a notification for the next non-empty bucket is needed, each individual message age timer in the buckets can be made "passive". More specifically, they can be passive timer in the past, which is simply a time stamp and does not need to be decremented at each time click. This represents a great savings in kernel time (considering the overhead of decrementing millions of timers per time click). Only one active timer is used, which is given the amount of time pertaining to that next bucket. STP creates this active timer as another node in the managed timer tree, and receives notification events from the event manager, along with those of the other types of timers.

There are both pros and cons for this bucket implementation. The advantage is that the frequent, per-packet timer update can be done in O(1) operation, which is to simply increase the time value and put it in the right bucket. The insertion sort complexity of managed timer is avoided. The disadvantage is that time precision is limited, depending on the number of buckets. Having more buckets increases the precision. Because STP is a control protocol, and it takes time to process and prepare BPDU packets anyway, time precision is not too important. Therefore we think pros outweigh cons for this approach.

3.5 List of available Spanning Tree extensions and features

This section aims to list the existing STP features which could sooner or later become part of ENA-STP as STP features or STP extensions.

3.5.1 802.1D core STP

This is the implementation of the STP core protocol, and will be available immediately. The core without an extension does not define any instance and therefore cannot perform any computation. The extensions create the connection between the interfaces and the core.

3.5.2 Uplinkfast

This feature enhances the STP core state machine by adding a fast transition from blocking to forwarding in some scenarios (refer to [3] for details). It will be implemented as part of a later implementation of the core (not in the first release).

3.5.3 Backbonefast

This features reduces the convergence time in case of failure by cutting the max age timer. It will be implemented in a later release of the core.

3.5.4 PVST+

This is the main extension available for STP. It provides a one-to-one mapping between an instance and a VLAN by creating loop free topologies on a per VLAN basis. This extension will be available together with the core in the first release.

3.5.5 MI-STP

MI-STP is a new extension of the STP core which provides a one-to-many mapping between one instance and some VLANs. This extension is not expected to be available in the first release, but depending on the time constrains it could be implemented later.

3.5.6 VLAN bridge

VLAN bridge is an STP extension defined to provide the loop free topology capability in a layer 3 environment by blocking a set of VLAN (and their respective SVIs) from receiving/transmitting some specific L3 traffic. This extension will be implemented later.

3.5.7 Portfast

In catOS, this has been considered a STP feature because it makes a port move to the FORWARDING state immediately after the link comes up. However in ENA portfast has been decoupled from STP and is now owned by the process responsible of the interface management (port manager for the catalyst 6000, see [4] for details).

3.5.8 BPDU guard

BPDU guard is a feature responsible to verify misconfigurations of interfaces configured with portfast enabled. It achieve this goal by checking whether a portfast enabled interface receives BPDUs (usually not expected). This feature will be incorporated into the core code later.

3.5.9 Root guard

Root guard is a feature responsible to guard the root switch from moving "on the other side" of a link. To check this kind of misconfigurations a perimeter of root-guard-enabled ports must be established by the network administrator; later this feature will ensure that the root switch is inside that perimeter and never moves outside. This feature will be implemented in the first release of the STP core.

3.5.10 Channel loop detection

This feature verifies inconsistencies in the PAgP configuration of a logical interface using BPDUs received. It will be implemented in the same time frame of PAgP (WHEN????)

3.5.11 Q-in-Q BPDU tunneling

This feature applied to an interface enables the interface to work with packets with a double 802.1Q encapsulation. From the STP perspective it requires the flooding of the doubly-encapsulated SSTP BPDUs received in order to avoid loops. This feature will be added later to the core.

4.0 Interactions with other components and features

Most of the interactions with other components will use the well defined mechanism provided by the sysdb facility. STP will be an EDM for a specific sub-tree of operational tuples (under /oper) and it will be a normal verifier for a sub-tree of configuration tuples (under /cfg). As STP is an EDM, a list of tuples for which we are going to provide notification must be declared in the documentation in order to avoid another component to wait forever for a notification it'll never get. Anyway the notification registration for an EDM-controlled sub-tree includes an initial request by sysdb to verify that a tuple is valid. For any tuple for which we are not going to give a notification, we will return a failure to sysdb. More details about generic sysDB tuples are provided in section 7.3, while all the platform dependent interactions with other components are provided in the platform dependent companion documents mentioned in section 1.

5.0 Memory and performance impact

The STP port database can easily become a very large database, depending on the number of extensions running at the same time and on the number of instances those extensions can enable. For this reason, the size of each extension should be tuned using the memory available on the specific platform where the protocol is implemented.

6.0 End user interface

This section lists all the user commands pertaining to spanning tree. They are divided into two categories: configuration commands and show commands. Configuration commands are used to set a value for spanning tree parameters. Show commands are used to display the current values.

6.1 Specifying spanning tree instances by *stp-list*

Most commands use the *stp-list* argument to specify the STP instance. The *stp-list* argument is to be entered as:

<ext-name *[inst-num-range]>*

where:

ext-name = { pvst+| mist| bridgegroup}

and inst-num-range is:

inst-num, [-inst-num]

If no *inst-num* is specified, the command applies to all instances for that extension.

Examples:

pvst– 3-10

mist 5-7, 9-12

bridgegroup 3-8

6.2 Configuration commands

Configuration commands are to be entered under <config> for instance-related commands, and <config-if> for port-related commands.

6.2.1 Enable/Disable STP

spanning-tree *stp-list*

no spanning-tree *stp-list*

Default

By default, STP is enabled.

Command Mode

Global configuration

Usage Guidelines

Implementation Notes

This command is related to loading and unloading of extension DLLs. By default pvst+ extension DLL is always loaded. mist and bridgegroup DLLs are loaded when the first instance is created, and are unloaded when the last instance is disabled.

Example

The following example shows how to enable spanning tree instances on mist 2 & 10.

```
router(config)# spanning-tree mist 2,10
```

Related Commands

6.2.2 Configure STP type

spanning-tree *stp-list* **protocol {ieee|dec|ibm|cisco|vlan-bridge}**

no spanning-tree *stp-list* **protocol**

A printed version of this document is an uncontrolled copy

Syntax Description

ieee	IEEE Ethernet Spanning-Tree protocol.
dec	Digital Spanning-Tree protocol.
ibm	IBM Spanning-Tree protocol.
cisco	A Cisco spanning tree protocol used on token ring CRF VLANs to prevent loops within a CRF (Concentrator Relay Function).
vlan-bridge	A Cisco spanning tree protocol to be used on bridge spanning tree instances that are bridging two or more VLANs together.

Default

The default protocol configured is **ieee** for pvst+ and mist instances, and **vlan-bridge** for bridgegroup instances.

Command Mode

Global Configuration Mode.

Usage Guidelines

cisco protocol is meant to be used on token ring CRF (Concentrator Relay Function) VLANs, which are not supported in the first release.

vlan-bridge protocol is meant to be used by a bridge that is bridging two or more VLANs together and the network design requires that the pvst+ and mist spanning trees not be collapsed into a single spanning tree. This is accomplished by encapsulating the STP BPDU in a 802.3 packet sent to a Cisco assigned address with a Cisco assigned SNAP HDLC. These BPDUs will be forwarded like regular multicast packets over the active topology and only the participating routers will handle. Additional, any IEEE BPDUs that are received on the bridge interfaces will be dropped.

Implementation Notes

Changing the protocol of a spanning tree will cause spanning tree parameters to be changed to default values appropriate for the specified protocol.

6.2.3 Configure the root bridge

spanning-tree *stp-list* **root primary [diameter** *hops* **[hello-time** *seconds*]]

no spanning-tree *stp-list* **root primary**

Syntax Description

diameter	Layer 2 network diameter
hello-time	Hello time in seconds

Default

Default is **no spanning-tree root primary**.

Command Mode

Global Configuration

Usage Guidelines

Use the **spanning-tree root primary** to force this switch to be root bridge for the specified spanning tree instances.

Use the no form of this command to restore to default configuration.

These commands should only be used on backbone switches.

Implementation Notes

This command configures the bridge to be the root bridge for an instance. The bridge priority will be modified from the default (32768) to a significantly lower value so that the bridge becomes the root. STP checks the bridge priority for the current root bridges. The bridge priority will be set to 8192 if the current root priority is larger than this value, and will be set to 1 less the current priority if not.

The **diameter** keyword is used to specify the layer 2 network diameter (maximum number of hops between any two layer 2 stations). STP will automatically pick up an optimal hello time, forward delay time, and maximum age time for a network of that diameter. You can use the **hello-time** keyword to override the automatically calculated hello time.

Example

```
router(config-t)# spanning-tree pvst+ 10 root primary
```

Related Commands

6.2.4 Configure the secondary root bridge

spanning-tree *stp-list* **root secondary [diameter** *hops* **[hello-time** *seconds***]]**

no spanning-tree *stp-list* **root secondary**

A printed version of this document is an uncontrolled copy

Syntax Description

diameter	Layer 2 network diameter
hello-time	Hello time in seconds

Default

Default is **no spanning-tree root secondary**.

Command Mode

Global Configuration

Usage Guidelines

Use the **spanning-tree root secondary** on switches that are desired to act as root switch should the preferred primary root fail.

Implementation Notes

When you configure a switch as the secondary root, the spanning tree bridge priority is modified from the default value (32768) to 16384 so that the switch is likely to become the root bridge for the specified instances if the primary root bridge fails (assuming the other switches in the network use the default bridge priority of 32768).

You can run this command on more than one switch to configure multiple backup root switches. Use the same network diameter and hello time values as you used when configuring the primary root switch

Example

```
router(config)# spanning-tree pvst+ 20 secondary
```

Related Commands

6.2.5 Configure bridge priority

spanning-tree *stp-list* **priority** *bridge-priority*

no spanning-tree *stp-list* **priority**

This command sets the bridge priority for an instance. The priority value will be from 0 to 65535. Use the no form of this command to restore to the default.

Syntax Description

bridge-priority	The lower the number, the more likely the bridge will be chosen as root. When the IEEE Spanning-Tree Protocol is enabled, number ranges from 0 to 65535 with steps of 4096 (default is 32768).

Default

The default configuration when the IEEE Spanning Tree protocol is enabled on the switch: 32768

Command Mode

Global Configuration

Usage Guidelines

Implementation Notes

Example

The following example shows how to set the spanning-tree priority to 8192

```
router(config)# spanning-tree pvst+ 20,100-102 priority 8192
```

Related Commands

6.2.6 Configure the hello time

spanning-tree *stp-list* **hello-time** *hello_time*

no spanning-tree *stp-list* **hello-time**

6.2.7 Configure the forward_delay time

spanning-tree *stp-list* **forward-delay** *forward_time*

no spanning-tree *stp-list* **forward-delay**

6.2.8 Configure the maximum aging time

spanning-tree *stp-list* **max-age** *max_age*

no spanning-tree *stp-list* **max-age**

A printed version of this document is an uncontrolled copy.

Syntax Description

Default

The default configuration is shown in the following table.

Table 1: STP Time Parameters

	Default (pvst+ & mist)	Default (bridgegroup)	range
Hello time	2	2	1-10 seconds
Forward delay time	15	20	4-30 seconds
Max-age time	20	30	6-40 seconds

Command Mode

Global Configuration

Usage Guidelines

The above commands set the hello time, forward delay time and maximum aging time for an instance.

6.2.9 Configure port priority for port priority sets

spanning-tree portprio {p1|p2} *priority*

no spanning-tree portprio {p1|p2}

Syntax Description

P1	Port priority set 1
P2	Port priority set 2

Default

The default configuration is for both **P1** and **P2** to have a priority value of 8 (priority is on 4 bits).

Command mode

Interface Configuration

Usage Guidelines

For each interface, STP assigns the individual port instance priorities into two priority sets, p1 and p2. Use this command to configure port priority values for priority sets. Use the no form of this command to restore port priority to default value 8.

Implementation Notes

Example

The following example shows how to set the spanning-tree port priority or priority set p1 to 10. Note that this command is not associated to any interface.

```
router(config)# spanning-tree portprio p1 10
```

Related Commands

spanning-tree *stp-list* **portprio {p1|p2}**

6.2.10 Configure port instance priority

spanning-tree *stp-list* **portprio {p1|p2}**

no spanning-tree *stp-list* portprio

A printed version of this document is an uncontrolled copy.

Syntax Description

p1	Port priority set 1
p2	Port priority set 2

Default

The default configuration is for all STP instances to be assigned to port priority set **p1**.

Command mode

Interface Configuration

Usage Guidelines

For each interface, STP assigns the individual port instance priorities into two priority sets, p1 and p2. Use this command to specify a list of STP instances to be assigned to a given priority set. Use the no form of this command to restore the assignment to default p1.

Implementation Notes

Example

The following example shows how to assign the pvst+ instance 100 to port priority set p1 on interface 3/5.

```
router(config)# interface fastethernet 3/5
router(config-if)# spanning-tree pvst+ 100 portprio p1
```

Related Commands

spanning-tree portprio {p1|p2} *portpriority*

6.2.11 Configure port cost for port cost sets

spanning-tree portcost {c1|c2} *portcost*

no spanning-tree portcost {c1|c2}

Syntax Description

c1	Port cost set 1
c2	Port cost set 2

Default

The default configuration is for both **c1** and **c2** to have a port cost value as listed in Table 2.

Command mode

Interface Configuration

Usage Guidelines

For each interface, STP assigns the individual port instance costs into two port cost sets, **p1** and **p2**. Use this command to configure port cost values for port cost sets. Use the no form of this command to restore port costs to default value.

Implementation Notes

Example

The following example shows how to set the spanning-tree port cost of port cost set c1 to 100. Note that this command is not associated with any interface.

```
router(config)# spanning-tree portprio c1 100
```

Related Commands

spanning-tree *stp-list* **portcost {c1|c2}**

6.2.12 Configure port instance cost

spanning-tree *stp-list* **portcost {c1|c2}**

no spanning-tree *stp-list* portcost

Syntax Description

c1	Port cost set 1
c2	Port cost set 2

Default

The default configuration is for all STP instances to be assigned to port cost set c1.

Command mode

Interface Configuration

Usage Guidelines

For each interface, STP assigns the individual port instance costs into two port cost sets, c1 and c2. Use this command to specify a list of STP instances to be assigned to a given port cost set. Use the no form of this command to restore the assignment to the default c1.

Implementation Notes

Example

The following example shows how to specify pvst+ instances 100-105 to have the port cost of port cost set p1 on interface 3/5.

```
router(config)# interface fastethernet 3/5
router(config-if)# spanning-tree pvst+ 100-105 portcost c1
```

Related Commands

spanning-tree portcost {c1|c2} *portcost*

Table 2: Default Spanning Tree Parameters

Parameter	**Default value**	**Range**
Bridge priority	32768	0-65535
Port priority	32	0-63
Port cost - Gigabit Ethernet	4	0-65535
Port cost - Fast Ethernet	19	0-65535
Port cost - Ethernet	100	0-65535
Port cost - ATM	14	0-65535

6.3 Show commands

6.3.1 The show spanning-tree command

A printed version of this document is an uncontrolled copy.

show spanning-tree *stp-list* **[interface** *interface-list*] **[active]**

show spanning-tree *stp-list* **[interface** *interface-list*] **summary**

These commands are used for displaying the spanning tree state information of each of the specified spanning tree instances. If the optional *stp-list* is omitted, the command will display information for all "active" instances in the system or, if an *interface-list* is specified, will display information for all the instances associated with the specified interfaces. If both a *stp-list* and *interface-list* are specified, the command will display information for those specified spanning tree instances to which the specified interfaces are associated.

```
router># show spanning-tree pvst+ 10
pvst+ is enabled for instance 10

Designated root data:
ID: Prio 32768 / SysExtID 10 / MAC 00-40-0b-8f-8b-ec
Max Age 20 sec   Hello Time 2 sec   Forward Delay 15 sec

Bridge data:
ID: Prio 32768 / SysExtID 10 / MAC 00-40-0b-8f-8b-ec
Max Age 16 sec   Hello Time 2 sec   Forward Delay 10 sec

Designated root reachability:
Root Port: none   Cost: 0

Interface     Vlan  Port-State   Cost  Port Prio (ID)
------------  ----  -----------  ----  --------------
FastEth1/1     10   Forwarding    80     10 (1A24)
```

6.3.2 The show spanning-tree statistics command

show spanning-tree *stp-list* **[interface** *interface-list*] **statistics**

This command displays the spanning tree statistical information for the specified port, as shown in the following example.

```
router># show spanning-tree pvst+ 10 interface ethernet 5/8 statistics
Spanning Tree enabled for pvst+ instance 10
     BPDU-related parameters
port spanning tree                     enabled
state           disabled
port id        0xcccf
port number        0x7eb
path cost    80
message age (port/VLAN)       0(10)
designated root                      00-10-2f-52-eb-ec
designated cost                           0
```

A printed version of this document is an uncontrolled copy

```
designated bridge                  00-10-2f-52-eb-ec
designated port       0xcccf
top_change_ack      FALSE
config_pending      FALSE

     PORT based information & statistics
config bpdu's xmitted (port/VLAN)     0(0)
config bpdu's received (port/VLAN)    0(0)
tcn bpdu's xmitted (port/VLAN)     0(0)
tcn bpdu's received (port/VLAN)     0(0)
forward trans count 0
...
```

6.4 Syslog messages

- All the syslog messages will be listed here.

7.0 Design considerations

From this section onward we'll focus on the design aspects of ENA-STP, given the functional model described in the previous sections.

7.1 Core database

The core database of STP is composed by ports (organized in instances) and by timers. As far as the ports database is concerned, we have two requirements to speed up the computation of the STP core, and these requirements must control the choice for the data structure we are going to use to maintain the STP port database. The requirements are as follows:

- The information for a specific port must be retrieved as quick as possible whenever an operation is requested for that port (a BPDU received and a port timer expired are the most critical cases)
- All the ports of an instance should be easily available for iteration in a "list" fashion, because all the topology recomputations involve a scan of all the ports of the instance.

The first requirement would lead us to choose a data structure with fast look up capability, where the look up must use the pair (instance ID, IF handle) as key[1]. Among the various choices, indexing is the fastest possible, as the time complexity is O(1). However the cardinality of the set of pairs (instance ID, IF handle) is extremely high, and grows as O(n*m) (whereas n is the cardinality of the instance IDs' set, and m is the cardinality of the IF handles' set). A more reasonable choice would be one of the many algorithms available which guarantee a logaritmic time for search and a linear space occupation.

The second requirement instead has its natural implementation in a set of linked lists organized on a per instance basis, while any other implementation would introduce higher overhead, from both the time and resources perspective.

Obviously these two requirements are conflicting, but we think they are so important to be worth a data structure with a limited amount of data redundancy. For this reason we are going to follow the model indicated in figure 6. When the cardinality of the Instance IDs' set is provided at run time (see section 7.3.1), an array (instance table) is created, allowing us to index an instance by its core instance ID. Each entry of the instance table contains instance specific data, plus two pointers, one referencing a doubly linked list of ports, the other

[1] The reason being that a port is an interface mapped to a specified instance.

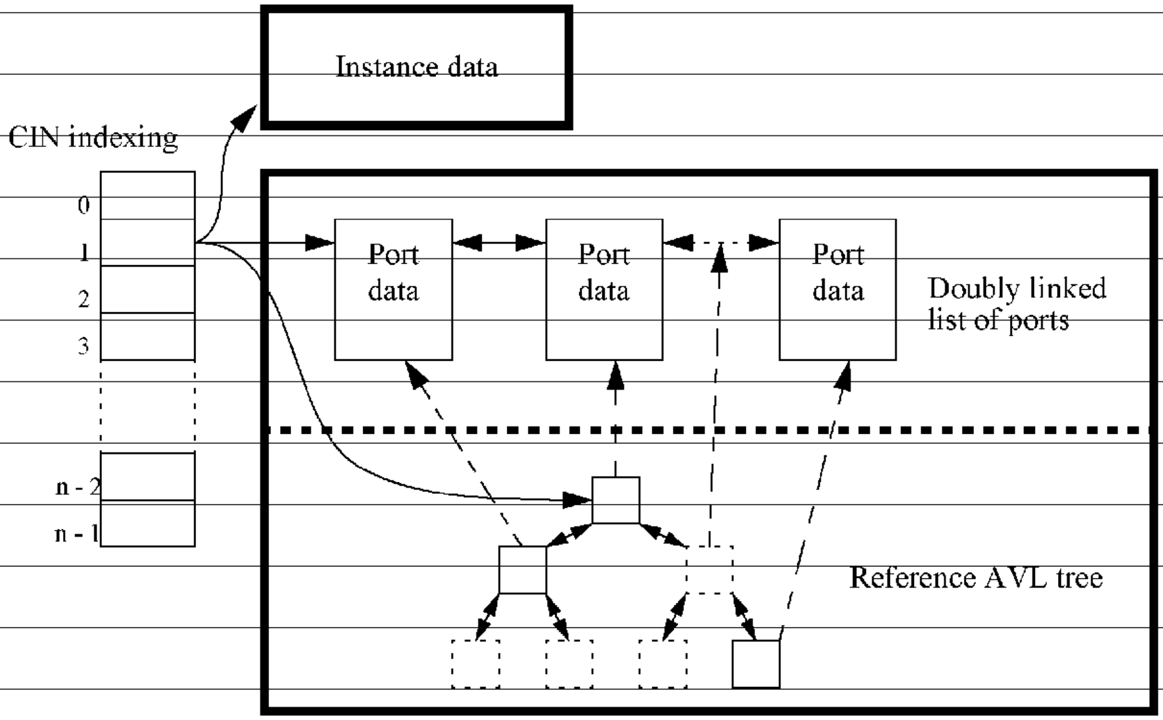

Figure 6, STP ports database

referencing an AVL tree. The doubly linked list is used to scan all the port data of an instance whenever the STP computation requires that; for this purpose only the forward pointer is used, while the back pointer is used only when a specific port data must be deleted. The AVL tree contains references to the port data, and is used only for the sake of quick look up of specific port data. A list of the properties of this database follows:

- Each element of the AVL tree is only referencing an element of the linked list, and this reference is unidirectional, i.e. the element of the linked list does not have a reference to the corresponding element of the tree.
- For this reason, the deletion process must always be started by a look up inside the AVL tree.
- The linked list does not need to be ordered in any specific way.

Each port data then is logically organized in such a way that some different sections can be identified, as shown in figure 7. The access control section is distributed in the head and tail of the data structure, in order to provide a simple magic-number mechanism for corruption detection. The head portion of the access control contains also the pointers that build the doubly linked list and a reference count, useful to keep track of the number of places where this port pointer is currently referenced (for deleting purposes).

The instance specific data instead are organized as shown in figure 8.

The access to the ports database can be done only for the following purposes:

- Update or read upon request of a specific port data pointer, using (instance ID, IF handle) as key.
- Scan of an entire instance ID through an iteration algorithm.
- Insertion and deletion of a port with key (instance ID, IF handle).

All the methods use well defined APIs, therefore the core code implementing the Spanning Tree algorithm will never need to worry about the specific implementation of the database.

7.1.1 Request of a specific port

The available APIs to request a specific port are:

```
cin = stp_get_cin(ein, extension_handle);
```

A printed version of this document is an uncontrolled copy

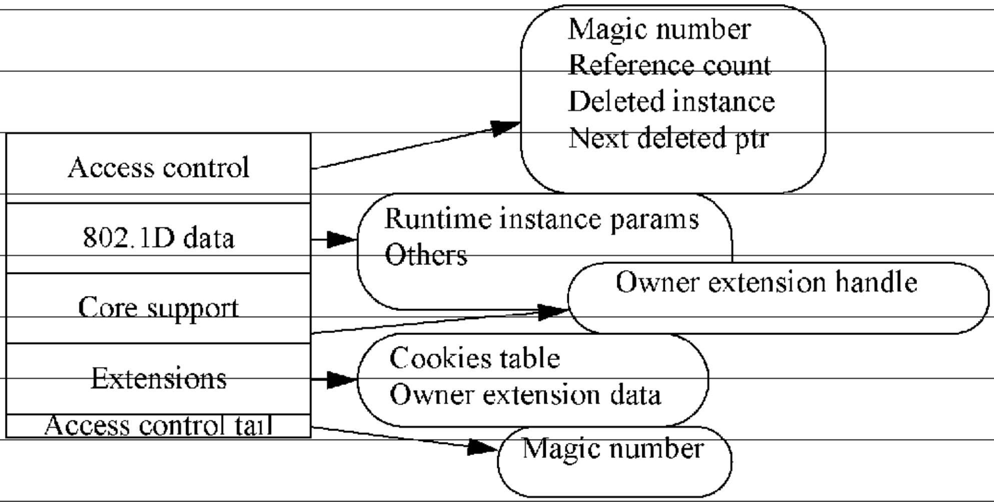

Figure 8, Instance data organization

```
cin = stp_get_cin(cin);
port_data = stp_get_pd(cin, if_handle);
port_owner_cookie = stp_get_port_owner_cookie(port_data,
extension_handle);
port_cookie = stp_get_port_cookie(port_data, extension_handle);
stp_release_pd(port_data);
```

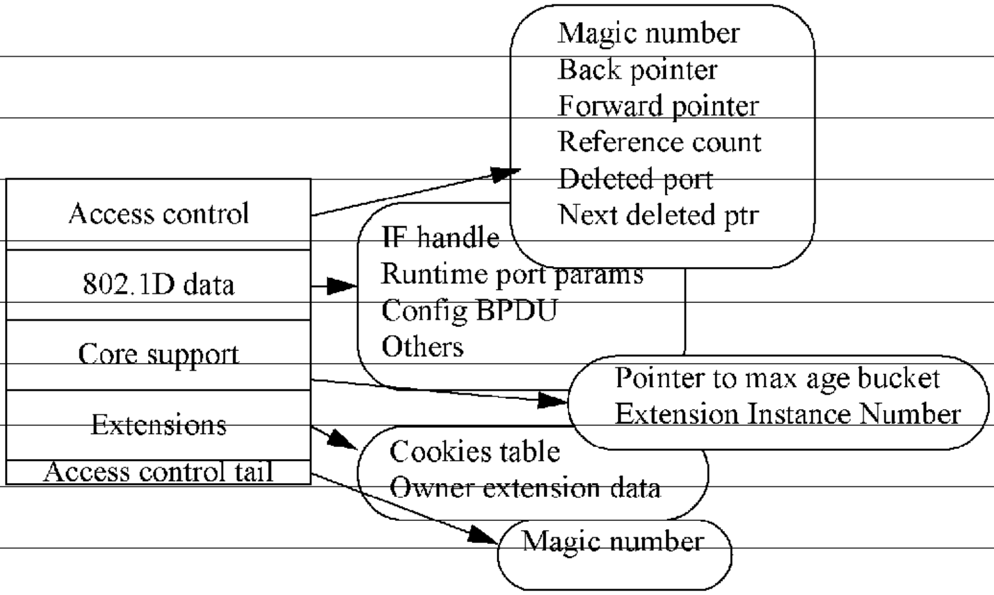

Figure 7, Port data organization

A printed version of this document is an uncontrolled copy

Every time you get a port, the reference count for that port is increased (port acquired) in order to avoid dangling pointers when a delete happens. For this reason each request of a port must be followed by a release, to decrement the reference count. The type of port_data provided by those functions does include only 802.1D data and core support; all the other info shown in figure 7 are hidden to the user and accessible only through a well defined interface. This is to prevent extensions from accessing directly to their cookies.

Note that there is no reference count associated with the cookies (and therefore no release operation necessary). In fact a cookie in a port can be removed for two reasons:

- because the associated port has been deleted
- because the associated extension has requested to remove all its cookies, typically as part of the cleanup before disabling the extension.

For the first case, the reference count of the port itself is of enough protection. For the second case, we can assume that any extension is implemented is such a way that no references to those cookies are kept by the extension at the time the request is done, and the core does never reference that.

7.1.2 Request of common data for a specific instance

We did not mention this in the previous sections, but each instance in the instance table does have per-instance data as well as the pointers to the AVL tree and the linked list for the ports. To access these data, an extension (or the core itself) can use the following APIs:

```
instance_data = stp_get_instance_data(cin);

instance_owner_cookie = stp_get_instance_owner_cookie(cin,
extension_handle);

instance_cookie = stp_get_instance_cookie(cin, extension_handle);

stp_release_instance_data(cin);
```

7.1.3 Iteration of an instance

```
stp_init_pd_iterator(&it, cin);

while(stp_iterate_pd(&it)) {

    port_data = stp_get_iter_pd(&it);

}

stp_close_pd_iterator(&it);
```

Iterating all the port_data in an instance allows the STP algorithm to perform most of the computation required. The initialization of the iterator requires an opaque pointer to a support structure for the iteration. After the initialization the iteration causes the list of ports to be scanned and the reference count of each port to be incremented/decremented accordingly without requiring any explicitly acquire/release of the port from the user. The current port under iteration can be retrieved using a specific API which gets the information from the opaque pointer used for the iteration. This scheme is “safe“ in the sense that a port_data can be deleted using the specific API (see section 7.1.4) while under iteration.

To make sure that nobody tries to iterate an iterator which has not been properly initialized, the opaque pointer has its own magic number that is set during the initialization and removed after the iterator has been closed.

7.1.4 Adding and removing ports

```
rc = stp_add_pd(port_data, cin);

rc = stp remove pd(port data);
```

The API to add a port to the STP database is pretty simple and adds the new port data to the head of the linked list; the AVL tree is updated accordingly.

The API to remove a port from the STP port database is far more complex. To ensure safeness and avoid dangling pointers, whenever a remove request comes, the API first detach the port data from the linked list and removes the node in the AVL tree referencing the entry. Then, if the reference count is different from zero, the function does not clear the forward pointer in the access control fields of the port, because the iteration can use it to proceed, and the port data is appended to a list of ports waiting to reach a reference count of zero; this list uses the "next deleted ptr" in the access control section. From now on, the port data will be waiting for all the release requests, and the last release request will free the memory and reorganize the list of deleted entries.

7.2 Handling cookies

7.2.1 Owner cookie

Each instance is associated with one extension. That extension is considered the owner of the instance. We expect that the owner of an instance wants to create a cookie only for instances it owns, not for all instances. The same applies to ports. For this reason we defined the owner cookie as something different from a normal cookie, which apply to any instance.

The owner cookies (instance and port) are automatically created by the core using the information inside the sysDB /cfg sub-tree.

7.2.2 Normal cookies

Each extension running in the process context of the STP core can request a cookie for its own data. The core provides two possible cookies to the extensions:

- instance cookie
- port cookie

An extension can request an instance cookie when it has some state information (some data) that is common for all the ports of every instance. It can request space for a port cookie when it needs to take a state on a per port basis.

The cookies are implemented as small tables of cookies' pointers (ToCP) in each port/instance. A ToCP can grow/shrink when extensions request to add/remove cookies. In this way, while the small ToCP can change its real location while growing/shrinking, the pointers to the real extensions data are always at the same memory locations.

Internally the core keeps track of which extension requested a cookie, by using two global tables, one containing the instance cookies' offset in each instance ToCP and another containing the port cookies' offsets in each port ToCP. Each extension is provided with an extension handle well defined at runtime (see section 7.3.1), so that the proper offset can always be reached using a single level of indirection. The functions to allocate and free a cookie are:

```
rc = stp_create_instance_cookie(extension_handle, cookie_size);

rc = stp_create_port_cookie(extension_handle, cookie_size);

rc = stp_remove_instance_cookie(extension_handle);

rc = stp_remove_port_cookie(extension_handle);
```

All the functions return an error code, and if the creation is successful. then all the existing ports or all the existing instances will have the required space for the cookie. It is responsibility of the extension to initialize that space in all the instances, as STP core will just allocate the memory an zero it. Some hooks are provided specifically for this purpose (see section 7.4).

A printed version of this document is an uncontrolled copy.

7.3 STP tuples in sysDB

The STP core alone cannot have any port attached, and therefore cannot define any instance; however it is able to work with the concept of instances, if somebody else provides the material to work on. In an object oriented environment, we could say that STP core defines an abstract base class with a certain amount of data associated with it, while each extension inherits from the base class and add some extra data to the data specific of the core. Whenever the core is involved in any computation for an instance defined as an object of a specific derived class (in our case, an STP extension), the core knows only (and works only on) the data specified by the base class.

In our sysDB environment, the implementation of this model requires each extension to define the tuple in an extension specific sub-tree, but make sure that each extension provides at least the tuples specified in the following sections.

7.3.1 The /cfg sub-tree

STP core uses normal sysdb tuples for the configuration sub-tree. A list of the most important tuples follows:

```
/cfg/stp/boot_ext/
```

Each extension can add a “directory” at this position. The directory is required to contain the following fields:

```
/cfg/stp/boot_ext/<ext_name>/enable
/cfg/stp/boot_ext/<ext_name>/init_dll
/cfg/stp/boot_ext/<ext_name>/ext_handle
```

Those three fields are used by the core to initialize any extension which is declared enabled; the initialization order is decided by the listing capability of sysDB: the core will simply do an iteration of “/ cfg/stp/boot_ext/*” and this should be taken into account. The core listen for notifications of any change under the /cfg/stp/boot_ext; in this way it can start or stop any extension based on a change of the “enable” flag or when a new directory is created. Obviously a change in the “init_dll” field does not have any effect until the next time the extension restarts. Instead any change to “ext_handle” should not be allowed unless “enable” is set to FALSE, and moreover the core must ensure that all the handles are unique among all the extensions. For this purpose the core is the verifier of /cfg/stp/boot_ext/<ext_name>/ext_handle.

```
/cfg/stp/boot_ext/<ext_name>/max_num_of_instances
```

This tells the maximum number of instances currently configured for an extension. The core allocates space for all the instances required by each extension (if that tuple is zero or not defined, then the core assumes the extension does not need any instance) by using a big array of instances. The core takes responsibility of providing a mapping between the internal flat instances (CIN) and the pair (extension, EIN). If this value is different from zero, then the extension is considered the owner of the set of instances created. If this is the case, few more tuples must be available for the purpose of building the owner cookies (as explained in section 7.2.1). Those tuples are:

```
/cfg/stp/boot_ext/<ext_name>/port_owner_cookie_size
/cfg/stp/boot_ext/<ext_name>/instance_owner_cookie_size
```

and contain the size in bytes of the owner cookies. This information is used after the initialization of the extension to create the instances.

```
/cfg/stp/ext/<ext_name>
```

Extensions can add their own local configuration in a specific subdirectory of this path. They could use / cfg/stp/boot_ext/<ext_name>/ for the same purpose, but as the core is registered for notification on that sub-tree, it is better to reduce the number of unnecessary notifications.

Other self explanatory tuples are reported in the tables below:

A printed version of this document is an uncontrolled copy

Table 3: Sysdb items in /cfg/stp/ext/<ext_name>/instance/<EIN>

SysDB item	Type	Values
hello_time	UINT8	1-10
forward_delay	UINT8	4-30
max_age	UINT8	6-40
priority	UINT16	0-65535
enable	UINT8	TRUE or FALSE

Table 4: Sysdb items under /cfg/if/<if>/stp/ext/<ext_name>

SysDB Item	Type	Values
port_cost_one	UINT16	0-65535
port_cost_two	UINT16	0-65535
port_cost_set	CLIENT	bitstream
port_prio_one	UINT16	0-63
port_prio_two	UINT16	0-63
port_prio_set	CLIENT	bitstream

7.3.2 The /oper sub-tree

STP core is an EDM for its operational sub-tree, with root in /oper/stp. Any extension interested in having operational tuples should intercept the sysDB requests.

Table 5: SysDB items under /oper/stp/ext/<ext_name>/instance/<EIN>

SysDB Item	Type	Values
hello_time	UINT8	1-10
forward_delay	UINT8	4-30
max_age	UINT8	6-40
designated_root_id	CLIENT	8 bytes of data
bridge_id	CLIENT	8 bytes of data
root_port_id	UINT16	2 bytes of data
topology_change_time	struct time_spec	time value
topology_change_count	UINT16	0-65535
spantreestats	struct STPSTATS	stp statistics value

A printed version of this document is an uncontrolled copy

Table 6: SysDB items under /oper/stp/ext/<ext_name>/instance/<instNo>/if/<if>

SysDB Item	Type	Values
port_id	UINT16	0-65535
portstats	struct PORTSTATS	stp port statistics value

Note that the port_id indicated in table 5 is just a copy of the port_id in /oper/if/<if>/port_id. This latter tuple is requested to be available for any interface in order to have a unique 12 bits identifier to be used inside the BPDUs for debugging purposes.

7.4 Hooking to the STP core

As already mentioned many times, the STP core provides to available extensions the capability to be hooked to some well known positions in the core functionalities, in order to modify the behaviour of the core algorithm. Each hook has a well known hook number associated to it. An hook number is an index into a table which addresses a list of function pointers with a well defined prototype and return type. Specifically, the return type can assume the following values:

```
STP HOOKRETURN DONE
STP_HOOKRETURN_BREAK
STP_HOOKRETURN_CONTINUE
```

Every time the core computation reaches an hook, the corresponding hook number is used to retrieve the list of functions that need to be called. The first function in the list is called, and based on it return value the core decides whether the next function should be called (STP_HOOKRETURN_CONTINUE) or the hook computation is complete (STP_HOOKRETURN_BREAK). If then the hook wants to completely replace the core for this functionality, it can return STP_HOOKRETURN_DONE (because STP_HOOKRETURN_BREAK does only stop the list walk). This allows a particular extension to filter a specific data from some other extension, but makes very important the order in which the functions are called. This order is provided by the extension handle.

Hooks can be added and removed by calls to:

```
stp_add_hook(hookNumber, extension_handle, func_ptr);
stp_remove_hook(hook_number, extension_handle);
```

Typically an hook is added at the initialization of the extension, and it is removed as part of the cleanup before the extension is "killed"; for this reason a special hook number STP_HOOK_ALL is provided to simplify this cleanup.

An initial list of hooks follows. The list could change during the implementation, based on specific requirements of the extensions:

STPHOOK_PREPARSEEVENT

STPHOOK_POSTPARSEEVENT

STPHOOK_MAPINSTANCE

STPHOOK_INITPORT (constructor)

STPHOOK_INITINSTANCE (constructor)

STPHOOK_CLEARPORT (destructor)

STPHOOK_CLEARINSTANCE (destructor)

A printed version of this document is an uncontrolled copy

STPHOOK_SENDBPDU

7.5 Receiving BPDUs

The reception of BPDUs is an event which cannot be handled by the core alone, unless an extension provides the redirection to the proper instance the BPDU belongs to. For this reason any extension responsible of instances and BPDUs should create an hook for STPHOOK_PREPARSEEVENT, and when the event is a BPDU, it should call the interface:

```
stp_process_bpdu(if_handle, cin, bpdu);
```

so that the core then knows what to do. The BPDU passed to the core through this API should be filtered of any extension specific field, i.e. it should be the BPDU with the format specified in the 802.1D standard starting from the "flags" fields; in fact, the core cannot check the BPDU version, as this is usually tight with the BPDU dialect understood by the extension only.

7.6 Extensions initialization, cleanup and sysDB identification

As mentioned in section 7.3.1, each extension can be enabled and disabled by configuring a specific tuple in sysDB. Whenever a /cfg/stp/boot_ext/<ext_name>/enable becomes TRUE, the core receives a notification from sysDB, and reacts to it by loading the DLL specified in /cfg/stp/boot_ext/<ext_name>/init_dll. This DLL must contain an entry point function:

```
stpext_init(extension_handle, ext_name);
```

This initialization function should take care of loading any other required DLL, setting up the cookies, adding all the hooks and registering a cleanup function to be called if and when /cfg/stp/boot_ext/<ext_name>/enable becomes FALSE. The extension handle and the extension name are input parameters to the DLL from the core, based on the contents of /cfg/stp/boot_ext/<ext_name>/ext_handle and /cfg/stp/boot_ext/<ext_name>. Moreover the core will keep internally the handle <-> name association to access to all the tuples required to build extension specific instances if /cfg/stp/boot_ext/<ext_name>/max_num_of_instances exists and it is different from 0.

The API available for registering the cleanup function is:

```
stp_register_cleanup(extension_handle, func_ptr);
```

Note that the extension handle is known to the extension itself thanks to the stpext_init() routine.

Reference Documents

[1] 802.1D

[2] "IOS/ENA STP cat6k extensions, Software Unit Functional/Design Spec," Marco Di Benedetto, Shawn Yang, ENG-61346

[3] "Spanning Tree Fast Uplink Switchover," Dinesh Dutt, ENG-14043

[4] "Catalyst 6000/ENA, Port Manager Functional/Design Specification," Chui-Tin Yen, Al Annaamalai, Mukundan Sudarsan, Simone Arena, ENG-51320